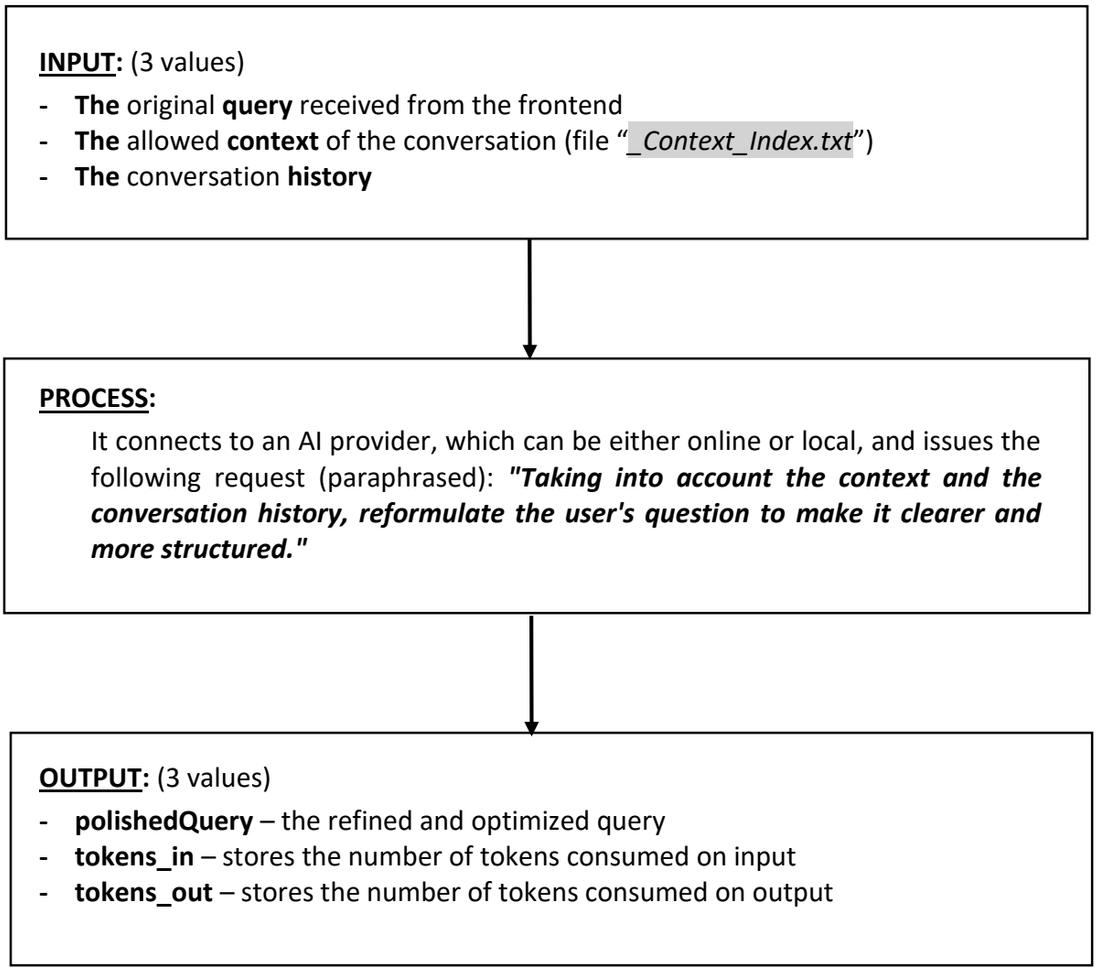
Section 5 – Main Flow of the Algorithm in smartFunctions.js

This section describes how the different SmartFunctions in the smartFunctions.js file work.

1. async function polishQuestion()

```
async function polishQuestion(originalQuery, allowedContext, history)
return { polishedQuery, tokens_in, tokens_out }
```

Description: This SmartFunction is designed to take the imperfect question the user adds to the conversation and return a perfect prompt. This prompt is essential to ensure the question is clear and understandable, while remaining within the context of the conversation and its history.



This function serves multiple purposes: **(1)** It allows users to make informal comments and exclamations without disrupting the conversation flow. **(2)** It ensures the context and history are correctly embedded into the prompt, eliminating the need to reintroduce the history in future steps of the algorithm—saving many tokens and avoiding redundant processing.

2. async function identifyFiles()

async function identifyFiles(polishedQuery, documentIndex, maxChunks)
return { relevantFiles, tokens_in, tokens_out }}

Description: With the optimized query, the algorithm searches for the most relevant files in the document index. This SmartFunction returns a list of chunks where the answer to the query should be looked for.

INPUT: (3 values)

- The **polishedQuery**, the perfect prompt used as the query
- The document index available in _Document_Index.txt
- The MAX_CHUNKS parameter defining the number of chunks to evaluate

PROCESS:

It connects to an AI provider, either online or local, and makes the following request (paraphrased): ***"The user has asked this question, and here is the index of the document's content divided into chunks. Based on the index, which chunks would you search in to find the answer to the user's question?"***

OUTPUT: (3 values)

- **Returns** relevantFiles, a string containing the names of the most relevant files where the answer to the query is likely to be found
- **tokens_in** – stores the number of tokens consumed on input
- **tokens_out** – stores the number of tokens consumed on output

3. async function buildAnswer()

async function buildAnswer(relevantFiles, queryText, maxAnswers)
return { finalAnswer, examinedChunks, positiveResponseCount, tokens_in, tokens_out };

Description: This function performs the search for answers to the query within each of the chunks listed by the identifyFiles() function. It combines the original question with the refined version (the perfect prompt) to enhance the search. The result may include one or more valid answers to the query.

INPUT: (3 values)

- **relevantFiles:** the list of chunks where the answers will be searched
- **queryText:** the combination of the original question + the polished query
- **maxAnswers:** the value of MAX_ANSWERS from the configuration

PROCESS:

It connects to an AI provider, which can be online or local, and issues the following instruction (paraphrased): ***"Answer questions strictly based on the document content. Also mention the index and/or Section and/or Legal Article and/or Chapter and verse, and include the direct quote. Do not infer, add, or assume anything. If no answer is found, reply with a single character: '-'."***

PROCESS:

This query process is repeated as many times as there are chunks selected for the search. The algorithm stops early if it reaches the maxAnswers threshold of positive results. Positive answers are concatenated into a finalAnswer, made up of all valid responses found.

OUTPUT: (5 values)

- **finalAnswer** the concatenation of all positive responses found
- **examinedChunks** the number of chunk-files actually queried
- **positiveResponseCount** the number of valid (positive) responses retrieved
- **tokens_in** – stores the tokens consumed on input
- **tokens_out** – stores the tokens consumed on output

This SmartFunction in the longest-running process in the algorithm, executing multiple AI queires. However, they are not resource-intensive in terms of processing of memory, since no embeddings are involved at any step of the operation.

4. async function polishAnswer()

```
async function polishAnswer(query, allowedContex, finalAnswer)
return { polishedAnswer, tokens_in, tokens_out }
```

Description: Since the previous function, buildAnswer(), may generate several valid responses to the same query, this function aims to deliver a single clear and well-organized answer.

Think of it like this: a student walks into a library with a written question and, after consulting several books, finds multiple valid answers. They then sit down to summarize everything into one coherent and concise answer. That's exactly what this final algorithm does.

INPUT: (3 values)

- **query**: the polished query is passed into this algorithm
- **allowedContext**: the algorithm ensures that the query remains within the allowed context of the conversation.
- **finalAnswer**: the combined set of all valid responses

PROCESS:

It connects to an AI provider -either online or local- and issues the following request (paraphrased):

"Organize and present the responses harmoniously as a single answer. Use Markdown format. Hightlight key ideas using bold text. Use bullet points and line breaks to improve readability. You may only respond based on the allowed context. Structure the response and do nt add anything beyond what's given. No autocompletions, opinions, or assumptions."

OUTPUT: (3 values)

- **polishedAnswer**: the final result – ths single, well organized response to be shown on the console of frontend.
- **tokens_in** – stores the number of tokens consumed on input
- **tokens_out** – stores the number of tokens consumed on output

Final Conclusion:

These are the four Smart-Functions that enable this algorithm to read unlimited documents, without requiring large-scale LLMs, embeddings, or training—delivering accurate answers with no hallucinations, at minimal cost.

The secret of the entire code is based on three core foundations:

(1) **The Concept Curve Paradigm:** It states that knowledge or a document should not be represented as a vector embedded in a space of hundreds of dimensions, but rather as a network of interconnected simple concepts.

(2) **The Concept Curve Embeddings Indexation (CC EI) Method:** A new way of indexing documents, which does not rely on compressing ideas into vectors with thousands of dimensions, but instead indexes documents as a set of interconnected simple concepts.

(3) **Smart-Functions:** These are AI-driven functions which, unlike OpenAI's function-calling, are provider-independent and can be executed on smaller models, even locally.

This algorithm is **highly cost-efficient** because it doesn't need advanced LLMs to operate—it works with lightweight models.

It is **computationally light** because it doesn't use embeddings, which would otherwise require heavy processing power to compress and compare vectors with thousands of parameters.

*It is **storage-friendly** because it doesn't store large multi-parameter vectors. Instead, it saves simple sets of words, which are essentially concepts representing text.*

For more information, consult **Agente CC**, a ChatGPT personality created to explain this method: https://tinyurl.com/agent-cc

The final section of the manual will teach how to perform indexation using the Concept Curve Embeddings Indexation method.

How is a Concept Curve Embeddings Indexation performed?

Just ask your LLM the following: *"Give me a group of 30 concepts that represent this chunk of text."*

That's it! That's how the indexation is done.

In the /data directory, where the document chunks are located, you'll also find the _Document_Index.txt file showing how CC EI indexation is implemented. More details will be published on the YouTube channel:

https://www.youtube.com/@Agente_Concept_Curve

That's all. Thank you very much.

Daniel Bistman